Design of the Shahkar Runtime Execution Environment Kit: *ShREEK*

Dave Evans
evansde@fnal.gov

24th September 2004

Contents

1 Introduction

Shahkar is a workflow management package used to run the software of multiple HEP experiments via an interpreted metadata based language. Executable Tasks are run in chains feeding the output of one stage into the input of the next, and each processing step needs to be monitored and checked upon completion to ensure data integrity. For this purpose, a structured runtime execution system is provided in the Shahkar package, which is intended to provide a common interface to monitoring information about the executable whilst camouflaging the details of the software from the execution site and its monitoring software. In addition, a generic monitoring framework with customisable plug-in dynamically loaded objects to interface to various monitoring systems is provided. The Shahkar subpackage that provides the core runtime utilities is called the *Shahkar Runtime Execution Environment Kit* or ShREEK .

2 The Main Script: Executor.py

The main ShREEK script is the Executor.py script that takes a list of executable tasks as produced by the Linker and executes the jobs in the list sequentially. The `Executor` exists for the entire lifetime of the job, and runs several services to provide information about the execution processes and an API to interact with them. The `Executor` is invoked via a wrapper script built and created by the batch interfaces used to create the job. Various options are available for the `Executor` object via a standard POSIX getopt command line mechanism. The command line options are listed in Appendix A. A broad outline of the functions performed by the `Executor` object are as follows:

- Load the set of tasks to be executed.
- Load the runtime configuration written out by the linker and ShREEK Interface and use it to initialise the various runtime tools required.
- Execute the tasks in a threaded subprocess.

- Provide event-driven and periodic monitoring of the execution process.
- Provide an XML-RPC [1] service on a TCP Port which can be used to interact with the job remotely.
- Handle any error conditions that arise during the processing in an intelligent way.

The `Executor` Object performs these tasks using a threaded execution model implemented with the python `threading.Thread`[2] class. The thread that handles the execution of the tasks is the `ExecutionThread` , while the `MonitorThread` handles the monitoring tasks. If an RPC service is required, it is implemented in a seperate thread. Each of these threads are described later in this document. A diagram showing the main components of the `Executor` and how they use the files produced by the Linker can be seen in Fig. 1.

3 Plugin Modules

Given the highly differing nature of output for different experiments executables, ShREEK provides a plugin mechanism that allows three catagories of service to be provided by the user and invoked at certain points within the execution process. These three catagories are:

1. **Updators** Functions which provide one or more fields of information to the `MonitorThread` MonitorState during periodic monitor updates. For example, updating the memory and CPU usage of a process.
2. **Monitors** Monitor Objects that provide handlers for certain events arising at various points in the job. For example, notifying a farm monitoring service of thet start of a new task or emailing users with a status update.
3. **ControlPoints** Control Points are invoked before and after executable tasks and allow responses to the job execution to customise execution flow. For example a control point may check for errors and attempt to re-run an executable that failed or run a rescue/cleanup task.

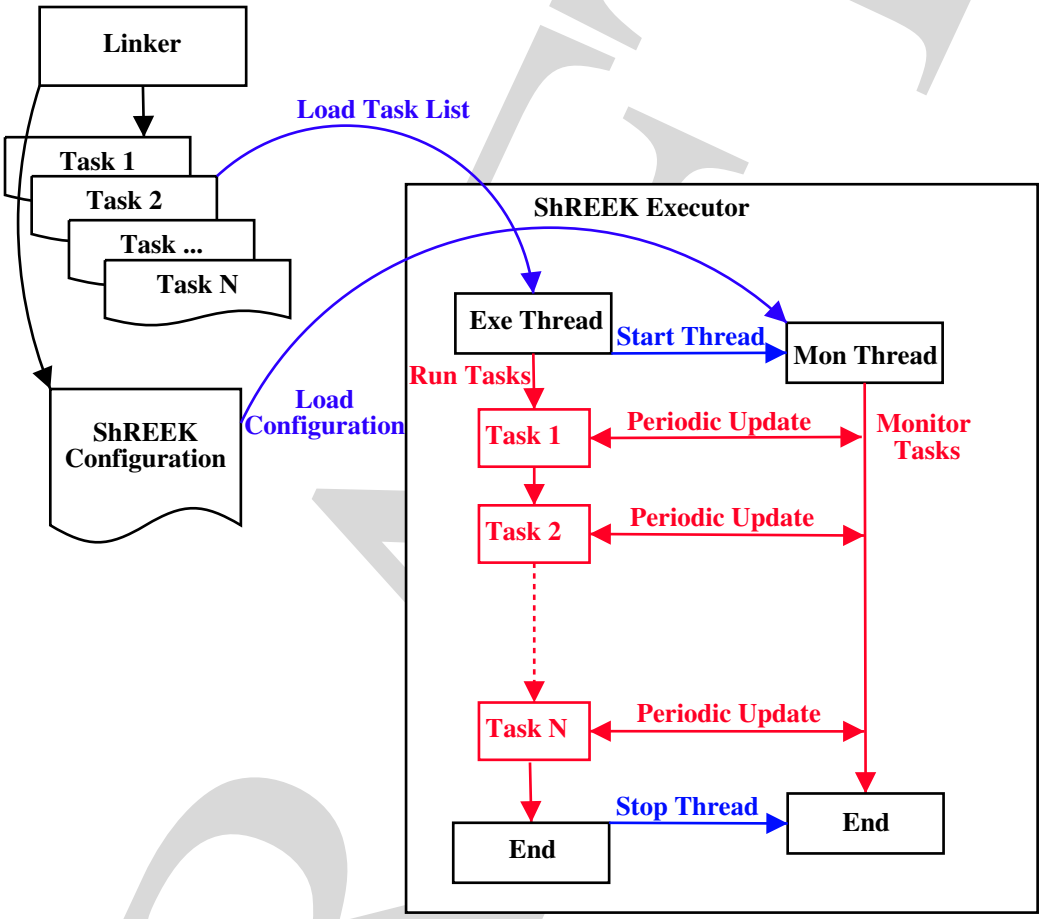

Figure 1: Diagram showing the main components of the ShREEK `Executor` Object. The Linker produces a task list and CfgFile in the job area which are loaded by the `Executor` . The tasks are then run in sequence by the `ExecutionThread` while being monitored by the `MonitorThread` .

ShREEK Plugins are python modules which must be on the PYTHONPATH at runtime. ShREEK provides a plugin registration facility that allows modules to register ShREEK plugins when they are imported. A list of plugin module names is given to ShREEK via its runtime configuration file and these modules are all dynamically imported, which triggers the plugin registration mechanism. The configuration file allows the user to the select and configure the various plugins they require for their execution task.

4 ShREEK Configuration

ShREEK is configured at controlled by a configuration object. This object is a python structure that contains information about how ShREEK will run. ShREEK Configuration objects can be saved and loaded into XML using the Shahkar XMLP package to save and load the object as an XML file. The information in the configuration object consists of the following:

- Tasks and execution order
- List of Plugin modules to be loaded
- List of Updators to be used to monitor tasks
- List of Monitors to be used and their configuration information
- List of Control Points and their configuration

The task configuration object can be created and managed using the `ShREEKInterface` Object and then saved into its XML form. This XML file is then passed to the ShREEK `Executor` via the command line interface *–taskcfg* option.

5 The `ExecutionThread` Object

The `ExecutionThread` Object runs the tasks in the sequence provided by the ShREEK Configuration and provides an API to the running tasks. The `ExecutionThread` provides an interface to send POSIX signals to the executing process in order to control it, and allow various monitoring systems

to interact with the task. The `ExecutionThread` performs the following steps during execution:

- Start the `MonitorThread` .
- Notify Start of job to `MonitorThread` .
- Loop through the list of executable tasks.
- Notify Start of each Task to `MonitorThread` .
- Changes the current working dir to the task subdir.
- Execute the task in a sub-process using a python `Popen4` Object [3].
- Poll the `Popen4` object until completion, or until a signal is recieved or error condition occurs.
- Changes the current working dir back to the main job dir.
- Notify End of each Task to `MonitorThread` .
- Notify End of job to `MonitorThread` .
- End the `MonitorThread` .

The `ExecutionThread` API for sending signals to the running task, as shown in Table 1.

Method	Description
killjob()	Terminate Execution of all tasks
killtask()	Send SIGTERM to current task sub process
suspendjob()	Send SIGSTOP to current task sub process
resumejob()	Send SIGCONT to current task sub process

Table 1: Table of signal API methods for the `ExecutionThread` Object. This set of methods can potentially be expanded to include all possible POSIX signals. Signals sent to the `ExecutionThread` are processed in a cyclical manner, via python `threading.Event` objects, and if an Event gets set, the appropriate signal is sent to the task process during the next cycle.

6 The MonitorThread Object

The MonitorThread is a periodically updating thread controlled by the ExecutionThread , that refreshes monitoring information. The updated information is distributed to a monitoring framework, containing adapter classes to handle certain monitoring conditions, by either acting on the information or forwarding it to some other source. This is achieved by creating an instance of a MonitorState Object and directing it at the currently executing task and periodically updating it. This object is then dispatched to the monitoring framework so that the information within it can be accessed by the monitor interfaces themselves. The MonitorThread also provides an API for notifying the monitor framework to other events such as change of tasks, signals, and error conditions. The update interval of the MonitorThread can be set with a command line option to Executor .

6.1 The MonitorState Object

The MonitorState Object provides a snapshot of an executable task by examining properties of the task as it runs. This information is used to proviude a snapshot of the task in terms of process parameters extracted from the Linux /proc filesystem and the incremental processing details such as event or filenumber. The MonitorState Object is derived from a subclass of python dictionary called UpdateDictionary which allows keys to be added along with a method to update the value of that key. An Update method calls the key specific monitor update methods for each key. Detailed documentation of the UpdateDictionary Object is provided in Appendix B.

The MonitorThread calls the Update method of the MonitorState and distributes the MonitorState object to each monitor adapter, where it can be used to extract information for distribution to monitoring systems.

The fields in the MonitorThread object are provided by ShREEK Updator Plugins, that can be registered from Plugin modules and selected for use in the configuration file. Updators are relatively simple tools that monitor one or two features of the job and add that information to the MonitorState .

7 XMLRPC Service

An XML-RPC interface to the monitor state can be run by adding the -r command line option for the `Executor` . This will start an HTTP based XMLRPC service on the port provided by the –portlist or –xmlrpcport command line options. The –xmlrpcport is used to specify a single port number to run the service on, the –portlist is used to provide a range of ports for when multiple jobs run on a single host. The default port used is 8080. The hostname of the port defaults to "localhost" and can be set via the –xmlrpchost option. When running the server makes the information in the `MonitorState` Object available via a remote service on the specified port. The accessor methods are defined in the RPCThread.py module, in the `RunjobRPCHandler` class. These methods can be called from XMLRPC clients and used to return the monitor information. The RPC service is implemented in a seperate thread via the `RPCThread` Object defined in RPCThread.py.

A Appendix A: Command Line Options to `Executor`

-m	Disable Periodic Monitor, shorthand for –noupdate
-t	Test Mode, shorthand for –test
-v	Verbose Mode, shorthand for –verbose
-e	Enable Exceptions: shorthand for –exceptions
-r	Run XMLRPC Server, shorthand for –xmlrpc
–xmlrpcport=	Port Number to run RPC interface server
–update=	Monitor Update period in seconds
–verbose	Verbose Mode
–noupdate	Disable Periodic Monitoring, Event driven monitoring still works
–tasklist=	Name of job list to be executed, default is TaskList.py
–exceptions	Allow Exceptions to be raised for development testing
–xmlrpc	Run XMLRPC Server for this job
–portlist=	List of port numbers to use for multi job nodes, comma sep arated list: 8080,8081,8082
–test	Test Mode, jobs are not executed
–tasklogfile=	Name of logfiles for redirecting stdout and stderr from tasks
–xmlrpchost=	Host name to use for RPC Service
–cfgkey=	Name Of Configuration Dictionary in tasklist module

B Appendix B: `UpdateDictionary` Documentation

`UpdateDictionary` inherits from the python dict type, and augments the functionality of the basic dictionary object by storing references to methods corresponding to certain keys. An Update Method is provided that calls the update method for each key that has one. Update Methods are called in the order that they are added to the dictionary. Normal keys without Update methods can be added in the usual way. The `UpdateDictionary` Interface is identical to that of the normal dictionary, with the only extra methods being for adding a key with an update method, and an Update method to call the registered methods.

- **AddUpdateKey(key,value,updator=None)**
  This method is used to add a key with an update method. The

method must be either a python `MethodType`, `FunctionType` or a callable `InstanceType`. The Update method is recorded for that key.

- **Update()**
  This method calls all the update methods for each key that has one.

Update methods that are registered must accept a single argument when called, which is the `UpdateDictionary` instance, so that other keys can be accessed by updator methods. The Updator method must return the new value to be stored under that key. An Example of using the `UpdateDictionary` object is provided below.

```
#  Define an update method
def Updator(updDict):
    #  return new value of key
    return "Updator Called"

ud = UpdateDict()
#  Normal Dictionary Keys can be added
ud['NormalKey'] = 'value'
#  Now add an updator key
ud.AddUpdateKey('Key1','Not Updated',Updator)
#  ud looks like:
#  {'NormalKey':'value','Key1':'Not Updated'}
#  Now call Update
ud.Update()
#  ud now looks like:
#  {'NormalKey':'value','Key1':'Updator Called'}
```

References

[1] XMLRPC Webpage.
`http://www.xmlrpc.com/`

[2] Python `threading` module `Thread` class.
`http://www.python.org/doc/current/lib/thread-objects.html`

[3] Python `popen2` module `Popen4` class.
`http://www.python.org/doc/current/lib/popen3-objects.html`